REMARKS/ARGUMENTS

Claims 1-3 and 5-22 are pending in the present application. Claims 1, 2, 5, 7, 8, 10, 11, 14, 17, 19 and 21 have been amended, and Claim 4 has been cancelled, herewith. Reconsideration of the claims is respectfully requested.

I. 35 U.S.C. § 101

The Examiner rejected Claims 1-22 under 35 U.S.C. § 101 as being directed towards non-statutory subject matter. This rejection is respectfully traversed.

In rejecting all of Claims 1-22, the Examiner states that:

> "The claims are *directed to a method* of executing a program asynchronously in distinct threads, but fail to indicate *how this method* of execution accomplishes a practical application. That is, the claims represent nothing more than a mere idea or concept, i.e. an algorithm, which is ineligible for patent protection. *Diamond v. Diehr*, 450 U.S. 175, 185 (1981)."

Applicants respond by showing that while Claims 1-9 are in fact method claims, the remaining Claims 10-22 are not method claims, and thus the Examiner's characterization regarding Claims 10-22 as being 'directed to a method' is clearly erroneous. Specifically, Claim 10 (and dependent Claims 11-13), Claim 14 (and dependent Claims 15-16) and Claim 17 (and dependent Claim 18) are directed to an apparatus, and not a method. For example, representative Claim 10 explicitly recites:

> "An apparatus for asynchronous execution within a program" (emphasis added)

Claims 10 and 14 are not directed to a mere idea or concept, as they expressly recite means for executing, *by the apparatus*, code in a first thread. Claims 10 and 14 are thus shown to not be a mere idea or concept, as they expressly recites *means for executing code by the apparatus*. Similarly, Claim 17 explicitly recites an interpreter that, upon detection of a keyword, creates a light weight thread and thus is not a mere idea or concept. It is thus urged that Claims 10-18 have been erroneously rejected under 35 USC 101 as being method claims that are directed to a mere idea or concept.

As to Claims 19-22, Applicants have amended such claims to comply with the statutory requirements of 35 USC 101.

Regarding method Claims 1-9, such claims have been amended to specifically recite code execution by processors within a multiprocessor data processing system, and thus such claims are not directed to a mere idea or concept but instead are specifically directed to processor execution of code.

Therefore, the rejection of Claims 1-22 under 35 U.S.C. § 101 has been overcome.

II. 35 U.S.C. § 103, Obviousness

The Examiner rejected Claims 1-22 under 35 U.S.C. § 103 as being unpatentable over Bonachea ("Bulk File I/O Extensions to Java"). This rejection is respectfully traversed.

Claims 1, 2, 9 and 19

The cited reference teaches *a series of asynchronous libraries that are called* to perform asynchronous I/O operations. The present invention is instead directed to a simpler and more elegant approach to enabling asynchronous operations to occur, through *use of inline keywords* that flag or direct that subsequent code be processed in a particular fashion during code execution – i.e. *a runtime determination*. The cited reference does not teach or otherwise suggest providing any such runtime determination capability, as the reference describes use of two compilers, a Titanium compiler and a C compiler (Section 2.2, second paragraph) which compile code and this code is then linked using a special library into executable code that is subsequently executed. *There is no ability to detect asynchronous flags during code execution.* The cited reference could not even be modified in accordance with Claim 1 without essentially gutting the essence, expressed purpose, and stated objectives of the teachings of the cited reference (a two-step compile, followed by a linking to special libraries to achieve high performance). The fact that a prior art device could be modified so as to produce the claimed device is not a basis for an obviousness rejection unless the prior art suggested the desirability of such a modification. *In re Gordon*, 733 F.2d 900, 221 USPQ 1125 (Fed. Cir. 1984). Although a device may be capable of being modified to run the way [the patent applicant's] apparatus is claimed, there must be a suggestion or motivation *in the reference* to do so. *In re Mills*, 916 F.2d 680, 16 USPQ2d 1430 (Fed. Cir. 1990). Because the overall architecture and implementation details are substantially different – the cited reference teaching a traditional compile/link/execute process, where any code particulars are determined during linkage and therefore are static in nature – there would have been no suggestion or other motivation to modify the teachings contained therein to provide a dynamic, run-time determination as to whether to execute an in-line code element in another thread based upon whether an in-line flag is encountered. *Importantly, the cited reference explicitly states that it does not support arbitrary multi-*

threading[1]. This further evidences that Claim 1 is not obvious in view of the cited reference, for if it was obvious, then Bonachea would not have expressly disavowed an ability to provide arbitrary multi-threading. The teachings of Bonachea, which are specifically directed to compile and link operations to provide asynchronous I/O operations, are not a mere design choice. Instead, this compile/link approach was taken because Bonachea did not know how to provide arbitrary multi-threading and thus did not know how to provide arbitrary asynchronous multi-threading. Thus, it is urged that Claim 1 has been erroneously rejected under 35 USC 103(a), as a proper prima facie showing of obviousness has not been established by the Examiner[2].

Claim 3

Applicants initially show error in the rejection of Claim 3 for similar reasons to those given above with respect to Claim 1.

Further with respect to Claim 3, such claim includes features that further emphasize the flexibility provided by the present invention. *The first keyword is usable both within a method (as shown at 404 of Figure 4A), as well as a type definition for a method (as shown at 414 of Figure 4B).* This flexibility in use advantageously provides that a block of code can include statements that are executed asynchronously with respect to the nesting level of that block, or alternatively to provide asynchronous processing at the block level if resource constrained (Specification page 12, last paragraph which extends to page 13). The cited reference does not teach or suggest either this claimed feature or its resulting advantages. Thus, it is further urged that Claim 3 has been erroneously rejected, as there are missing claimed features not taught or suggested by the cited reference.

Claims 5, 14 and 21

Applicants initially show error in the rejection of Claim 5 (and similarly for Claims 14 and 21) for similar reasons to those given above with respect to Claim 1.

Further with respect to Claim 5 (and similarly for Claims 14 and 21), such claim recites both a first keyword and a second keyword. The first keyword indicates a code element that may be executed out of order. The second keyword indicates that execution of the code element in the second thread (as

---

[1] As per Section 2.2 of Bonachea, it states "The only major Java feature which is not currently supported by Titanium is arbitrary multi-threading. That is, the user *cannot* create arbitrary asynchronous threads of control within a process" (emphasis added by Applicants).

[2] To establish prima facie obviousness of a claimed invention, all of the claim limitations must be taught or suggested by the prior art. MPEP 2143.03 (emphasis added by Applicants). *See also, In re Royka*, 490 F.2d 580 (C.C.P.A. 1974). If the examiner fails to establish a prima facie case, the rejection is improper and will be overturned. *In re Fine*, 837 F.2d 1071, 1074, 5 USPQ2d 1596, 1598 (Fed. Cir. 1988).

executed by the encountered the first keyword) *must complete before the next code element is executed.* Thus, these two keywords synergistically co-act to provide an ability to resynchronize code execution without polling for completion, as was done in the past. In fact, that is exactly how the cited reference achieves synchronization – by polling. As can be seen at Section 4.1.3 of the cited reference, "Done" methods are defined to achieve synchronization, and these "Done" methods check the status of the AsyncFileRequests the application is querying and returns a Boolean flag indicating whether the "Done" condition was satisfied. Thus, this "Done" routine is a polling routine that is used to determine if a previously dispatched task has completed – in other words, it is looking at the status of *past actions*. In contrast, Claim 5 is directed to a *forward-looking methodology*, where the second keyword indicates that *execution of the code element in the second thread must complete before the next code element immediately following the second keyword is executed.* It is, in effect, a brake that is applied on the next code element as can be seen by the delayed 'stuff2' execution shown in Figure 4E. **The teachings of the cited reference do not teach or otherwise suggest any type of keyword that *directly effects code execution of immediately following subsequent code*, but instead merely provides a polling method to determine whether a previously dispatched task has completed.** This can also be seen by the fact that a timeout parameter is also passed to the Done routine, and the Done routine returns when either (1) pending operations complete, or (2) the timeout has expired (whichever occurs first). The results of the Done routine is a true or false indicator indicating which of these two scenarios (1) and (2) actually occurred, and thus the processing immediately following the Done routine would have to query this true/false indicator to determine what action to take. Therefore, the execution of the next code element immediately following the Done routine is not conditioned upon completion of code in another thread (per Claim 5, "the second keyword indicates that *execution of the code element in the second thread must complete before the next code element immediately following the second keyword is executed*"). Rather, per the teachings of the cited reference, the next code element immediately following the Done routine is in all likelihood a test to determine whether the condition code provided by the Done routine is True or False to see whether the Done routine timed-out. Thus, it is shown that Claim 5 has been erroneously rejected under 35 USC 103(a), as there are missing claimed elements not taught/suggested by the cited reference.

Still further with respect to Claim 5, since the cited reference teaches a simple polling technique that merely looks back at previously dispatched tasks, and is not forward looking or otherwise forward-deterministic, it is shown that the cited reference does not teach the claimed step of "executing the next code element *in the first thread* after execution of the code element *in the second thread* completes" as there is no type of thread coordination provided by the simple "Done" polling technique that is provided by the teachings of the cited reference. Thus, Claim 5 is further shown to have been erroneously rejected

under 35 USC 103(a), as there are additional missing claimed elements not taught/suggested by the cited reference.

Claims 6, 15 and 22

Applicants initially show error in the rejection of Claim 6 (and similarly for Claims 15 and 22) for similar reasons to those given above with respect to Claims 1 and 5.

Further with respect to Claim 6 (and similarly for Claims 15 and 22), Applicants show that such claim recites a feature of "determining whether a third keyword *exists in the code element*, the third keyword indicating a statement that may be executed out of order". Thus, the code element for which out of order execution has been detected (by the first keyword) has, itself, a keyword that indicates a statement may be executed out-of-order, thus advantageously providing *a nesting capability for out-of-order execution*. The cited reference does not teach or otherwise suggest any such nested out-of-order execution. This nesting capability advantageously provides that a block of code can include statements that are executed asynchronously with respect to the nesting level of that block, as well as having other blocks of code nested within such block of code, to thereby provide recursive asynchronous execution of such block of code (Specification page 12, last paragraph). The cited reference does not teach/suggest either this claimed feature or its resulting advantage. Therefore, it is further urged that Claim 6 is not obvious in view of the cited reference as there are missing claimed features not taught/suggested by such cited reference.

Claim 7

Applicants initially show error in the rejection of Claim 7 for similar reasons to those given above with respect to Claim 1.

Further with respect to Claim 7, Applicants urge that the cited reference does not teach or suggest the claimed feature of "wherein the method is executed by a run-time interpreter". Rather, the cited reference teaches use of two compilers – a Titanium compiler and a C compiler – where code is compiled and then linked using a special library to achieve its stated performance objectives. Because of this compiler processing methodology, there is no ability to determine, *during code execution*, whether a first keyword exists in the code, the first keyword being a flag indicating that a subsequent code element following the first keyword may be executed out of order. Importantly, the cited reference expressly teaches away from using any type of run-time interpreter as the techniques are not compatible with such a run-time environment. Therefore, it is further urged that Claim 7 is not obvious in view of the cited

reference as there are missing claimed features not taught/suggested by such cited reference – such reference expressly teaching away from use of such claimed features (Bonachea Section 2.2, second paragraph).

Claim 8

Applicants initially show error in the rejection of Claim 8 for similar reasons to those given above with respect to Claim 7.

Further, Applicants urge that the fact that a prior art device could be modified so as to produce the claimed device is not a basis for an obviousness rejection unless the prior art suggested the desirability of such a modification. *In re Gordon, supra.* The cited reference teaches a technique for reducing latencies in I/O operations – and hence is directed to a technique for making software run faster in order to enable high-performance scientific applications (Section 1, Introduction; Section 2.2, Titanium). For example, the cite reference explicitly states:

> "This research focuses on maximizing the I/O performance for each node given a fixed workload." (emphasis added by Applicants)

The cited reference teaches use of two different compilers and associated compiling of code to achieve its performance objectives – the Titanium compiler performs various optimizations using knowledge of the parallel flow control, and then translates programs entirely to C where they are further optimized during a second compile operation (Section 2.2, second paragraph). Modifying the teachings of the cited reference in accordance with Claim 8 would in fact result in these teachings *failing to meet their stated objectives*, as a Java Virtual Machine interpreter is intrinsically slower in operation than the environment as taught by the cited reference. In addition, the current state of such JVMs do not provide optimization using knowledge of parallel flow control – an expressed required feature of the teachings of the cited reference. Thus, a person of ordinary skill in the art would not have been motivated to modify the teachings of the cited reference in accordance with the teachings of the claimed invention as the expressed desires and purposes of the cited reference would have been unachievable – strongly evidencing no motivation to modify such teachings in accordance with Claim 8. It is therefore shown that there was no suggestion of any desire to modify the teachings of the cited reference to include a Java Virtual Machine interpreter, and thus the basis of this obviousness rejection is shown to be error, per *In re Gordon, Id.*

Further, because there was no suggestion of any desire to modify the teachings of the cited reference in accordance with Claim 8, the only motivation for such modification must be coming from Applicants' own patent specification, which is improper hindsight analysis. It is error to reconstruct the

patentee's claimed invention from the prior art by using the patentee's claims as a "blueprint". *Interconnect Planning Corp. v. Feil*, 774 F.2d 1132, 227 USPQ 543 (Fed. Cir. 1985). Thus, Claim 8 is still further shown to have been erroneously rejected using impermissible hindsight analysis.

This failure by the reference to provide any suggestion to modify the teachings therein in accordance with Claim 8 can further be seen by the fact that the cited reference *expressly teaches away* from using an interpreter such as a Java Virtual Machine (JVM) due to resulting performance inadequacies (Section 2.2). This further evidences improper hindsight analysis being used by the Examiner in rejecting Claim 8, as the only motivation for making this modification in rejecting Claim 8 must be coming from Applicants' own patent specification, which is improper hindsight analysis. Thus, Claim 8 is further shown to have been erroneously rejected under 35 USC 103(a).

Claims 10, 11 and 16

Applicants initially show error in the rejection of Claim 10 (and similarly for Claims 11 and 16) for similar reasons to those given above with respect to Claim 1.

Further with respect to Claim 10 (and similarly for Claims 11 and 16), such claim recites that the *first keyword*, which is used to indicate an out-of-order capability, *is a type definition for a code element*, which advantageously allows for changing the actual declaration of the code element itself, as shown in Applicants' preferred embodiment in Figure 4B and described in the Specification at page 11, last paragraph. In contrast, per the teachings of the cited reference, a special purpose library is provided having unique names for individual methods that are invoked to perform asynchronous I/O operations. Quite simply, there is no teaching of a *type definition* for a subsequent code element that indicates an out-of-order execution capability for such subsequent code element. Thus, it is urged that the features of Claim 10 are not taught or suggested by the cited reference, and therefore is it shown that Claim 10 has been erroneously rejected under 35 USC 103(a).

Claim 12

Applicants initially show error in the rejection of Claim 12 for similar reasons to those given above with respect to Claim 10.

Further with respect to Claim 12, such claim includes features that further emphasize the flexibility provided by the present invention. The first keyword is usable both within a method, as shown at 404 of Figure 4A, as well as a type definition for a method, as shown at 414 of Figure 4B. This flexibility in use advantageously provides that a block of code can include statements that are executed asynchronously with respect to the nesting level of that block, or alternatively to provide asynchronous processing at the block level if resource constrained (Specification page 12, last paragraph which extends

to page 13). The cited reference does not teach or suggest either this claimed feature or its resulting advantage. Thus, it is further urged that Claim 12 has been erroneously rejected, as there are missing claimed features not taught or suggested by the cited reference.

Claim 13

Applicants initially show error in the rejection of Claim 13 for similar reasons to those given above with respect to Claim 10.

Applicants further show error in the rejection of Claim 13 for similar reasons to those given above with respect to Claim 4.

Claim 17

With respect to Claim 17, such claim recites the claimed feature of "wherein the interpreter, upon detecting the keyword, creates a light weight thread and executes the code element in the light weight thread". As can be seen, a light weight thread is *created by the interpreter* upon detecting the keyword, and the code element executes in the light weight thread. The cited reference does not teach an interpreter that creates a light weight thread upon detecting a keyword, as expressly recited in Claim 17. Because of the substantial architectural differences between the teachings of the cited reference and the invention recited in Claim 17, there is no ability of an interpreter to create a light weight thread upon detecting a keyword, and then executing code in the created thread due to use of a compiler/linker that generates machine executable code that is subsequently executed by a processor. In rejecting Claim 17, the Examiner asserts that the cited reference teaches this claimed feature of a thread creation by an interpreter at section 4.1.2. Applicants urge that this cited passage is directed to details of I/O initiation methods, and when these methods are called, they perform type-checking, bounds-checking and end-of-file checking, and then initiate the requested asynchronous I/O operation. There is no indication that this asynchronous I/O operation is executed in a newly created thread that is created by an interpreter itself. Thus, it is urged that Claim 17 has been erroneously rejected under 35 USC 103(a) as there are missing claimed features not taught or suggested by the cited reference.

Claim 18

Applicants initially show error in the rejection of Claim 18 for similar reasons to those given above with respect to Claim 17.

Applicants further show error in the rejection of Claim 18 for similar reasons to those given above with respect to Claim 8.

Closing Statement

Finally, in closing, the Examiner alleges that the reference states "that any Java environment could be used" (page 8 of the most recent Office Action dated 3/7/06), when in fact the reference states "It is our belief that the concepts explored and performance results are relevant to *any high-performance dialect of Java that wishes to support the I/O demands of array-based, data intensive applications*" (emphasis added by Applicants). "Any Java environment" (as characterized by the Examiner) is very different from "any high-performance dialect of Java that wishes to support the I/O demands of array-based, data intensive applications" (as stated in the reference). Such broad-brushed mischaracterization of the teachings of the cited reference is clearly improper, and in error. *The reference explicitly acknowledges that its techniques are not compatible with an arbitrary multithreading environment* (Section 2.2, third paragraph), and the features of the presently claimed invention(s) are specifically directed to such a multithreaded environment.

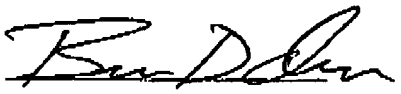
Therefore, the rejection of Claims 1-22 under 35 U.S.C. § 103 has been overcome.

III. Conclusion

It is respectfully urged that the subject application is patentable over the cited reference and is now in condition for allowance. The Examiner is invited to call the undersigned at the below-listed telephone number if in the opinion of the Examiner such a telephone conference would expedite or aid the prosecution and examination of this application.

DATE: June 2, 2006

Respectfully submitted,



Brian D. Owens
Reg. No. 55,517
Wayne P. Bailey
Reg. No. 34,289
Yee & Associates, P.C.
P.O. Box 802333
Dallas, TX 75380
(972) 385-8777
Attorneys for Applicants